

# Path O-RAM: An Extremely Simple Oblivious RAM Protocol

Emil Stefanov  
UC Berkeley  
emil@cs.berkeley.edu

Elaine Shi  
UC Berkeley  
elaines@cs.berkeley.edu

## Abstract

We present Path O-RAM, an extremely simple Oblivious RAM protocol.

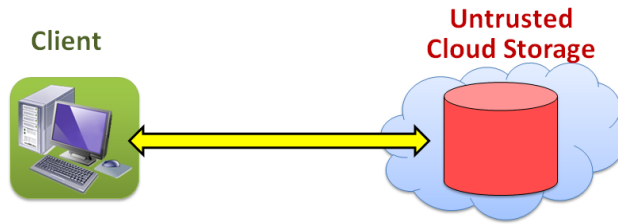
## 1 Introduction

In this technical report, we present an extremely simple construction for Oblivious RAM with low-bandwidth overhead. We call our protocol *Path O-RAM*.

## 2 Related Work

Oblivious RAM was first investigated by Goldreich and Ostrovsky [4, 5, 10] in the context of protecting software from piracy, and efficient simulation of programs on oblivious RAMs. Since then, there has been much subsequent work [1–3, 6–9, 11–15].

## 3 Problem Definition



**Figure 1: Oblivious RAM scenario.**

We consider a client that wishes to store data at a remote untrusted server while preserving its privacy (depicted in Figure 1). While traditional encryption schemes can provide confidentiality, they do not hide the data access pattern which can reveal very sensitive information to the untrusted server. In other words, the blocks accessed on the server and the order in which they were accessed is revealed. We assume that the server is untrusted, and the client is trusted, including the client’s processor, memory, and disk.

The goal of O-RAM is to completely hide the data access pattern (which blocks were read/written) from the server. From the server’s perspective, read/write operations are indistinguishable from random requests.

**Notations.** We assume that the client fetches/stores data on the server in atomic units, referred to as *blocks*, of size  $B$  bytes each. For example, a typical value for  $B$  for cloud storage is 64 KB to 256 KB. Throughout the paper, we use the notation  $N$  to denote the capacity of the O-RAM (i.e., the total number of data blocks that the O-RAM can hold).

**Simplicity.** Our construction is extremely simple in contrast with previous constructions. It consists of only 18 lines of pseudo-code as shown in Figure 2.

**Security definitions.** We adopt the standard security definition for O-RAMs from [14]. Intuitively, the security definition requires that the server learns nothing about the access pattern. In other words, no information should be leaked about: 1) which data is being accessed; 2) how old it is (when it was last accessed); 3) whether the same data is being accessed (linkability); 4) access pattern (sequential, random, etc); or 5) whether the access is a read or a write.

**Definition 1** (Security definition). *Let*

$$\vec{y} := ((\text{op}_1, u_1, \text{data}_1), (\text{op}_2, u_2, \text{data}_2), \dots, (\text{op}_M, u_M, \text{data}_M))$$

*denote a data request sequence of length  $M$ , where each  $\text{op}_i$  denotes a  $\text{read}(u_i)$  or a  $\text{write}(u_i, \text{data}_i)$  operation. Specifically,  $u_i$  denotes the identifier of the block being read or written, and  $\text{data}_i$  denotes the data being written.*

*Let  $A(\vec{y})$  denote the (possibly randomized) sequence of accesses to the remote storage given the sequence of data requests  $\vec{y}$ . An O-RAM construction is said to be secure if for any two data request sequences  $\vec{y}$  and  $\vec{z}$  of the same length, their access patterns  $A(\vec{y})$  and  $A(\vec{z})$  are computationally indistinguishable by anyone but the client.*

Like all other related work, our O-RAM constructions do not consider information leakage through the timing channel, such as when or how frequently the client makes data requests.

## 4 The Path O-RAM Protocol

### 4.1 Overview

We first give an overview of the Path O-RAM protocol. The client stores a small amount of local data in a cache. The server-side storage is treated as a binary tree where each node is a bucket that can hold up to a fixed number of blocks. We maintain the invariant that at any time, each block is mapped to a uniformly random leaf bucket in the tree, and uncached blocks are always placed in some bucket along the path to the mapped leaf. Whenever a block is read from the server, the entire path to the mapped leaf is read into the cache, the requested block is remapped to another leaf, and then the path is written back to the server. When the path is written back to the server, additional blocks in the cache may be evicted into the path as long as the invariant is preserved and there is remaining space in the buckets.

## 4.2 Server Storage

**Definition 2** (Tree). *The server stores a binary tree data structure of height  $L$  and  $2^L$  leafs. The tree can easily be laid out as a flat array. The levels of the tree are numbered 0 to  $L$  where level 0 denotes the root of the tree and level  $L$  denotes the leafs.*

**Definition 3** (Bucket). *Each node in the tree is called a bucket. Each bucket can contain up to  $Z$  blocks. If a bucket has less than  $Z$  real blocks, the remaining blocks are dummy blocks. The entire bucket is always stored as an encrypted fixed-size byte array, and hence the server can't distinguish between real and dummy blocks.*

In practice, we would choose  $Z$  to be a small number such as 3 or 4.

## 4.3 Client Storage

**Definition 4** (Cache). *At any point in time, the client stores a small subset of the  $N$  possible blocks into a data structure  $C$  called the cache.*

**Definition 5** (Position map). *The client stores a position map array  $\text{position}[u]$  that consists of  $N$  integers, mapping each block  $u$  to one of the  $2^L$  leaf buckets in the server's tree data structure. The mapping is random and hence multiple blocks may be mapped to the same leaf and there may exist leafs to which no blocks are mapped. The position map changes over time as blocks are accessed and remapped.*

## 4.4 Initialization

The client cache  $C$  is initially empty. The position map initially contains `null` for the position of every block. The position `null` is a special value indicating that the corresponding block has never been accessed and the client should assume it has a default value of zero.

## 4.5 Reads and Writes

**Definition 6** (Path). *The path  $\mathcal{P}(p)$  is the set of identifiers of tree buckets along the path from the root of the tree to the leaf bucket with index  $p$ .  $\mathcal{P}(p, \ell)$  is the identifier for the bucket in  $\mathcal{P}(p)$  at level  $\ell$  in the tree.*

The protocol for reading/writing a block is described in Figure 2. The algorithm can be summarized in four simple steps:

1. **Remap.** Randomly remap the position of block  $u$ . (Lines 1 to 2)
2. **Read.** Read the path containing block  $u$ . (Lines 3 to 5)
3. **Update.** If the access is a write, update the data stored for block  $u$ . (Lines 6 to 11)
4. **Write.** Write the path back and possibly include some additional blocks from the cache. (Lines 12 to 17)

If the special case that the requested block has a position of `null` (i.e., it has never been read before), the client should access a random path and place a zero block in the O-RAM for the corresponding position.

**Access**(op, u, data\*):

```

1:  $p \leftarrow \text{position}[u]$ 
2:  $\text{position}[u] \leftarrow \text{UniformRandom}(0 \dots 2^L - 1)$ 
3: for  $\ell \in \{0, 1, \dots, L\}$  do
4:    $C \leftarrow C \cup \text{ReadBucket}(\mathcal{P}(p, \ell))$ 
5: end for
6:  $\text{data} \leftarrow \text{data}'$  for which  $(u, \text{data}') \in C$ 
7: if op = write then
8:    $C \leftarrow C - \{(u, \text{data})\}$ 
9:    $\text{data} \leftarrow \text{data}^*$ 
10:   $C \leftarrow C \cup \{(u, \text{data})\}$ 
11: end if
12: for  $\ell \in \{L, L - 1, \dots, 0\}$  do
13:    $S \leftarrow \{(u', \text{data}') \in C : \mathcal{P}(p, \ell) = \mathcal{P}(\text{position}[u'], \ell)\}$ 
14:   Remove  $\min(|S|, Z)$  random blocks from  $S$ .
15:    $C \leftarrow C - S$ 
16:    $\text{WriteBucket}(\mathcal{P}(p, \ell), S)$ 
17: end for
18: return data

```

**Figure 2: Algorithm for data access.** Read or write a data block identified by  $u$ . If op = read, the input parameter  $\text{data}^* = \text{None}$ , and the Access operation returns the newly fetched block. If op = write, the Access operation writes the specified  $\text{data}^*$  to the block identified by  $u$ .

## 5 Extensions

### 5.1 Integrity

Our protocol can be easily extended to provide integrity (with freshness) for every access to the untrusted storage. Because data from untrusted storage is always fetched and stored in the form of a tree path, we can extend the tree data structure stored on the server to become a Merkle tree. Hence the client only needs to store a single hash of the Merkle tree root in order to verify the integrity and freshness of any path.

## 6 Experiments

We conducted experiments to measure the size of the client's cache for different bucket sizes and number of levels. Figures 3 and 4 show the results.

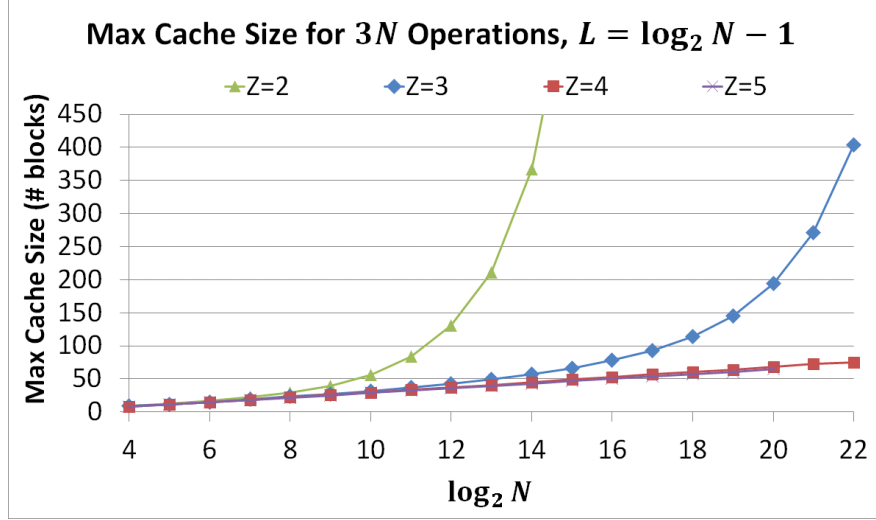


Figure 3: Client cache size for different bucket sizes (denoted as  $Z$ ).

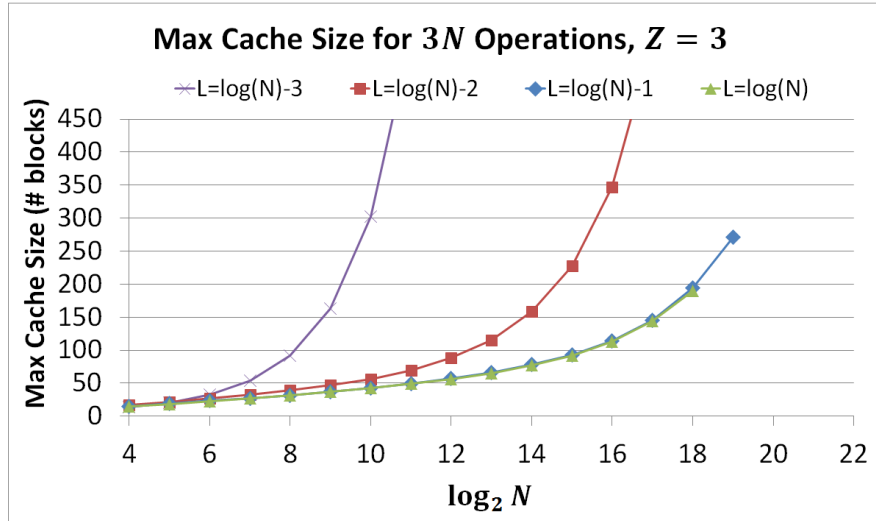


Figure 4: Client cache size for a different number of levels in the tree (denoted as  $L$ ).

## References

- [1] P. Beame and W. Machmouchi. Making rams oblivious requires superlogarithmic overhead. *Electronic Colloquium on Computational Complexity (ECCC)*, 17:104, 2010.
- [2] D. Boneh, D. Mazieres, and R. A. Popa. Remote oblivious storage: Making oblivious ram practical. Manuscript, <http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf>, 2011.

- [3] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious ram without random oracles. In *Proceedings of the 8th conference on Theory of cryptography*, TCC'11, pages 144–163, Berlin, Heidelberg, 2011. Springer-Verlag.
- [4] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, 1987.
- [5] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 1996.
- [6] M. T. Goodrich and M. Mitzenmacher. Mapreduce parallel cuckoo hashing and oblivious ram simulations. *CoRR*, abs/1007.1259, 2010.
- [7] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pages 95–100, New York, NY, USA, 2011. ACM.
- [8] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 157–167. SIAM, 2012.
- [9] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 143–156. SIAM, 2012.
- [10] R. Ostrovsky. Efficient computation on oblivious rams. In *STOC*, 1990.
- [11] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 294–303, New York, NY, USA, 1997. ACM.
- [12] B. Pinkas and T. Reinman. Oblivious ram revisited. In *CRYPTO*, 2010.
- [13] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with  $o((\log n)^3)$  worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.
- [14] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious ram. NDSS, 2012.
- [15] P. Williams and R. Sion. Usable PIR. In *NDSS*, 2008.